METHODOLOGY FOR TESTING AND VALIDATING KNOWLEDGE BASES

C. Krishnamurthy, S. Padalkar, and J. Sztipanovits

Center for Intelligent Systems

Vanderbilt University, Nashville, Tennessee

B.R. Purves

Boeing Aerospace Company, Huntsville Alabama

ABSTRACT

The paper describes a test and validation toolset developed for artificial intelligence programs. The basic premises of this method are: (1) knowledge bases have a strongly declarative character and represent mostly structural information about different domains, (2) the conditions for integrity, consistency and correctness can be transformed to structural properties of knowledge bases and (3) structural information and structural properties can be uniformly represented by graphs and checked by graph algorithms. The interactive test and validation environment have been implemented on a SUN workstation.

INTRODUCTION

Testing and validation is the ultimate precondition for the application of artificial intelligence (AI) technology in space systems. In spite of its obvious significance, testing and validation have been a neglected topic in AI research. The results being reported are quite contradictory. Some authors have pointed out that certain knowledge-based systems, such as expert systems, are inherently untestable and unreliable, while others argue that software validation is easier for knowledge-based systems than for conventional programs.

The first section of this paper summarizes our results in the evaluation of AI technology from the aspect of software engineering. An important conclusion of this analysis is that clear separation between AI systems (expert systems, natural language systems, etc.) and AI techniques (declarative programming, symbolic programming, etc.) is necessary. It has been shown, that the well-known difficulties in testing and validation are inherent nature of the functionality of specific AI systems and do not stem from the implementation technology. Most importantly, the basic AI techniques offer new opportunities in software testing and validation, which can dramatically improve the test technology of complex software systems.

The second section of the paper describes a test and validation toolset developed for AI programming. The basic thrusts of the selected methodology are: (1) knowledge bases have strongly declarative character and represent mostly structural information about different domains, (2) the conditions for integrity, consistency and correctness can be transformed to structural properties of knowledge bases and (3) structural information and structural properties can be uniformly represented by graphs and checked by graph algorithms.

An interactive test and validation environment has been implemented on SUN workstation. The knowledge representation paradigms for which test and validation methods have been developed include: rule-based systems and object-oriented programming. The application of the methodology is presented for testing structural properties of object-oriented programs.

BACKGROUND

The problems of testing and validation can be examined only in the context of the system to be tested and validated. Therefore, clear distinction must be made between systems that are built using AI and the techniques developed and used in AI programming.

1. AI Systems and AI Techniques

One of the widely accepted, generic objectives of AI is to construct intelligent agents (Newell, 1982). Intelligent agents can operate autonomously in a task environment, are able to recognize their situation by means of the perceptual components, and are able to plan their actions according to a goal structure by means of their general knowledge. These capabilities are also manifestations of human intelligence, i.e., the primary objective of AI systems is to mimic human intelligence.

The AI systems which have received the largest publicity in recent years are expert systems. Their primary purpose is to represent human knowledge symbolically and "operate" on the knowledge by using automated reasoning methods. Some of the most important aspects of expert systems that have attracted considerable attention are:

- ability to capture rare and expensive human expertise and make it available,
- ability to reliably operate in fuzzy, unexpected situations,
- ability to implement heuristics,
- ability to explain actions for users.

While seeking a better understanding of human intelligence and implementing systems that exhibit "intelligent" behavior, research in AI has discovered a number of novel software techniques and tools. These techniques and tools have proven to be extremely useful in a number of application domains struggling with construction of highly complex systems. More importantly, AI techniques have provided methods to use computers for symbolic, qualitative "computations," which have the immediate potential for building new generations of application systems in areas such as instrumentation and process control. The approach, which focuses primarily on AI techniques and not so much on the scientific objectives of AI, (i.e., understanding and imitation of human intelligence) is often referred to as AI engineering (Allmendinger, 1986).

It would be difficult to enumerate all of the new software techniques originated and elaborated by AI research. Here we discuss only declarative programming, which is widely used in the implementation of intelligent systems.

Conventional programming is essentially imperative, i.e., programs describe the sequence of steps that are necessary for solving a particular problem. We may state that imperative programs primarily represent "how to" knowledge. In imperative programming the programmer is responsible for transforming the problem definition ("what to") into its solution of imperative style.

Declarative programs describe the declarations of problems rather than their solution. The basic technique used in declarative programming is to build "smart" interpreters that can transform the declarations into "how to" knowledge. The key components of declarative programming are (1) the problem-specific representation language, which is used for describing the problem and (2) the corresponding interpreter.

Well-known programming paradigms that are strongly declarative are:

- logic programming, where programming occurs in the form of declaring objects and their relations, (a well known example of logic programming languages is, of course, Prolog),
- rule-based programming, where the knowledge is expressed primarily in rule format (e.g., ART, KEE, etc.),
- constraint-based programming, which includes the declaration of objects (e.g., variables) and the constraints (e.g., arithmetic

constraints) among them.

Declarative programming is widely used in constructing knowledge-based systems. The "knowledge base" is usually the declarative component while the interpreter is the procedural component of these systems (e.g., the rule base is the knowledge base, the inference engine is the interpreter in the case of rule-based expert systems).

2. Testability in AI Programming

Whether we approach AI programming from the side of specific AI systems (e.g., expert systems) or from the side of AI programming techniques (e.g., declarative programming), we can identify significantly different views concerning testing and validation.

From a functional point of view, expert systems try to mimic human expertise. The basic conceptual and practical problems stemming from this fact are clearly described by Lane, 1986.

a. Testing requires design specifications. Lane's observation is that specifications for expert systems, against which system performance can be evaluated "are almost universally lacking in current expert system developments." The probable reason is that though the concept of expertise is intuitively clear, it is impossible to give a unique specification for it (at least presently or in the immediate future). Obviously, the "rule-set" of rule-based expert systems can be considered only as a "model" of expertise, rather than its specification. He suggests the development of new methods for setting design requirements and system specifications that should be based on an improved understanding of the roles of expert systems in complex systems.

b. Performance is dependent on the scenario. A well-known problem of current and near-future expert systems is that their performance degrades dramatically at the "boundary of their knowledge base." Contrary to human experts, expert systems are unable to detect their limits so as to avoid catastrophic failures and to degrade gracefully in new or marginal conditions. Lane points out that except in the relatively simple cases, when the "expert system" is actually the implementation of a well-defined decision tree, the performance evaluation of expert systems has an "inherent dilemma." A possible method of testing is to sample the scenarios and conditions, and evaluate the system performance in specific situations. This method can fail to detect even potentially catastrophic outcomes. The other alternative is systematic enumeration of all possible input conditions, which is unrealistic in most cases due to time and cost.

Test approaches can help in the development of expert systems, but cannot resolve the problems mentioned above (Gashing et al., 1983).

The declarative character of the knowledge bases offers new opportunities for testing some of their structural and logical features. Validation methods are presented in Stachowitz et al., 1987; Nguyen, 1987; and Suwa et al., 1982; for checking inconsis-

tency, completeness, redundancy, etc., of rule bases. It should be mentioned that these tests cannot guarantee functional correctness but can offer significant help in detecting potential problems.

GENERIC TEST AND VALIDATION METHODOLOGY FOR KNOWLEDGE BASES

The basic thrust of our methodology is that the primary implementation technique for knowledge-based systems is declarative programming. As we have previously discussed, declarative programming includes three different program components, which are:

- interpreter,
- typically small imperative components, and
- declarations.

The interpreter and the imperative components are basically conventional programs that can be tested and evaluated by using well elaborated software engineering methods and techniques. In this sense, testing and evaluation of declarative programs does not differ from that of the conventional programs. The major difference is that the complexity of declarative programs is mostly concentrated in the declarations constituting the "knowledge base" of the system to be tested. Below we summarize some of the new opportunities emerging for testing and validation of declarative programs.

1. Automatic Proof of Correctness

Declarative programs are typically symbolic representations of structures. It is possible to implement automatic reasoning processes that can prove various properties of the structures represented. Requirements, such as:

"The fan-out must be less than or equal to 20," or
"Two active outputs cannot be connected"

can be easily checked on the declarative representation of a digital circuit simulator program. In other words, the functional correctness of the simulator can be tested by using automatic, high-level tools.

2. Mathematical Modelling

The structure of declarative programs can be mapped into graphs and different structural properties can be checked by using graph algorithms. E.g., causal networks which are used in failure mode and effect analysis can be tested for cycles; physical structures can be tested for connectivity; signal-flow structures can be tested for loops, etc. Graph algorithms can be used for testing the equivalence of different declarative programs, which is a unique possibility. (Proving the equivalence of imperative programs is an extremely complicated problem.)

3. Graphic Tools

Since declarative programs typically represent structures, they can be represented by graphic tools, and can be synthesized by inter-

active graphic editors.

Although, these opportunities have been recognized and exploited in some of the test and validation techniques mentioned before, their common feature is that the actual implementation is closely coupled to a particular knowledge-based system and knowledge representation language (Stachowitz, 1987).

Our goal was the development of a generic methodology and programming environment which effectively supports the testing and validation of different kinds of knowledge-based systems. The rationale behind this goal is the recognition that knowledge-based systems include multiple knowledge bases and are described in different representation languages.

The generic test and validation method can be summarized as follows.

Let us suppose, that L is the representation language and P is a set of declarations written in L. The general steps of validating the knowledge base are the following:

Specification of test criteria. By analyzing the specific nature of the knowledge base, a relevant set of test criteria [c(1), c(2),...,c(n)] has to be defined. The individual test criteria should be assertions on the structural properties of the knowledge base.

Specification of mapping rules. Depending on the semantics and syntactics of L, and the way the test criteria can be expressed as abstract graph properties, mapping rules (M) are defined. The rules maps P into a labelled, directed graph M(P)->G(V,E). The labels of the vertices and edges of the graph: v[a(1), a(2),...,a(n)] and e[a(1),a(2),...a(j)] are attributes that are extracted from P and associate the nodes and edges with its semantic entities.

Specification of user interface. The actual test proceeds by mapping the knowledge base (or certain sections of the knowledge base) into graphs and checking the test criteria by running graph algorithms. The results of the tests are presented by using a knowledge base specific graphic interface.

STRUCTURE OF THE TEST AND VALIDATION ENVIRONMENT

The methodology described above makes it possible for the design of a test and validation environment (TVE) where the common components are clearly separated from those which are unique to specific knowledge bases. The ultimate benefit of this separation is that the system can be easily adapted to different problems and representation languages and can provide a unified environment for testing and validating knowledge bases.

The structure of the TVE can be seen in Figure 1. The MAPPER accepts the knowledge base to be tested from the user and maps it into a graph. The ANALYZER runs a set of graph algorithms and outputs the results to the user. The analysis process is interactive and supported by graphics. The ANALYZER KERNEL constitutes the common part of TVE. It provides a set of

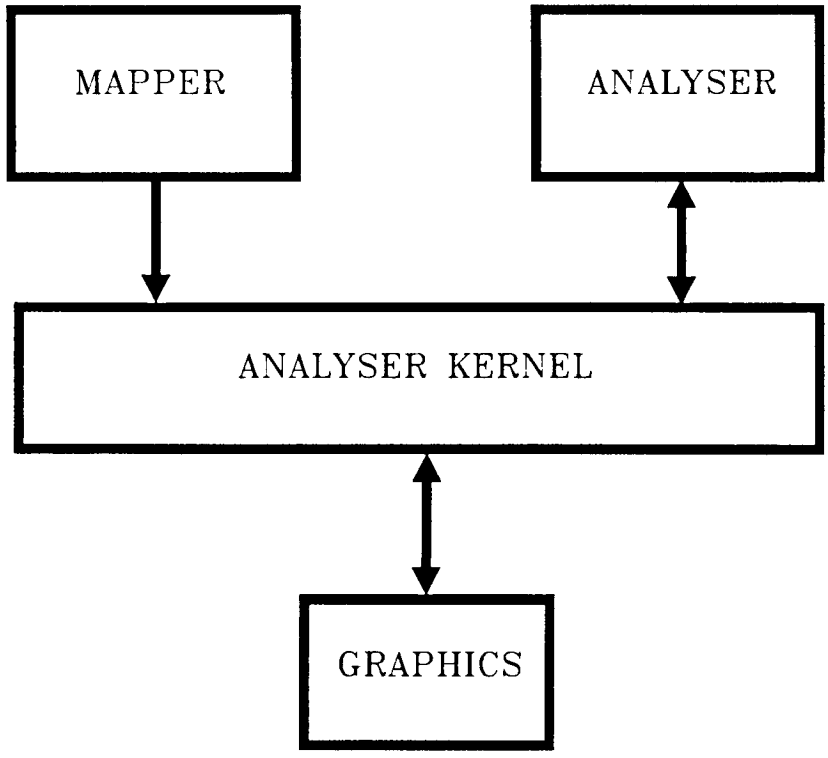

Figure 1: Functional Structure

services to build, represent and analyze graphs. The summary of the interfaces of the analyzer kernel can be seen in Tables 1, 2, and 3.

The Mapper Interface includes two sets of calls. One of them is used to parse the input source file which contains the knowledge base. The other set is used to create and modify graphs. The Analyzer Interface provides access to the library of graph algorithms which are the basic building blocks for implementing test and verification procedures.

The selection of graph algorithms is continuously expanded as new testing and validation methods are developed for different knowledge bases.

The third group of kernel calls facilitates generation of user interfaces. In order to help the user in navigating through complex structures and in analyzing structural properties, extensive color graphics are used with a sophisticated window system. The services provided by the graphics interface are summarized in Table 3. The interactive graphics interface makes it possible (1) to represent the entire graph, (2) to zoom into certain areas, (3) to select nodes and edges by using a pointing device and to display the corresponding semantic entity of the knowledge base in a text window, and (4) to start various analysis processes through a hierarchically organized menu interface.

IMPLEMENTATION

TVE has been implemented on a SUN 3/110 workstation by using the Sunview graphics package. The system is decomposed into two communicating processes (see Figure 2). The Analyzer and Mapper functions run as a LISP process. The appropriate kernel interface functions are written in C and are embedded in the LISP environment. The advantage of this solution is that the knowledge base specific components of the Analyzer and Mapper can be more conveniently implemented in LISP than in other available languages.

The graphics interface runs as a separate graphics process which communicates with the LISP process through UNIX pipes. After receiving a user command, it is decoded and the appropriate function call is sent to the LISP process to service the request.

Separation of the graphics interface from the other components of the system ensures the portability of TVE to other workstations, with different graphics capabilities.

APPLICATION EXAMPLE: TESTING AND VALIDATION OF OBJECT-ORIENTED SYSTEMS

Object-oriented programming has the virtue that hierarchical system declarations and properties, such as structural and functional inheritance, can map quite naturally into this programming methodology.

Typically, most of the useful object-oriented systems tend to become very large and, after a point, manual structural testing becomes extremely difficult, if not impossible. The TVE provides an automated, interactive test environment, with extensive graphics support, for the structural

Table 1. Mapper Interface

FUNCTION	PROCEDURE CALLS	DESCRIPTION
Parse input	[Internal set of macros specific to the representation language]	Builds symbol tables, stores text information
Create graph	create-node(attributes) create-edge(attributes)	creates a list of nodes edges, and graph adjency lists

Table 2. Analyzer Interface

FUNCTION	PROCEDURE CALLS	DESCRIPTION
Detect cycles	cycles (graph)	finds node-chains which form cycles in the graph
Find connected components	find-connected-components (graph)	finds a spanning forest for the graph
Find nodes matching certain attributes	find-group (graph attributes)	partitions the graph based on specific attributes of nodes or edges
Describe node	display-node (node)	displays all attributes of a node
Access nodes	gen-lower-tree(node) gen-upper-tree(node)	generates sub-tree rooted at this node

Table 3. Graphics Interface

FUNCTION	PROCEDURE CALLS	DESCRIPTION
Menu-based input	[executive calls]	converts analysis requests from graphics process into analyzer function calls
Graph layout generators	hierarchy (graph root), bipartite (graph) tree (graph root)	draws nodes and edges on screen
Highlight sections of graph	highlight (path graph) [executive calls]	highlights cycles, displays text, zooms on sections of the graph

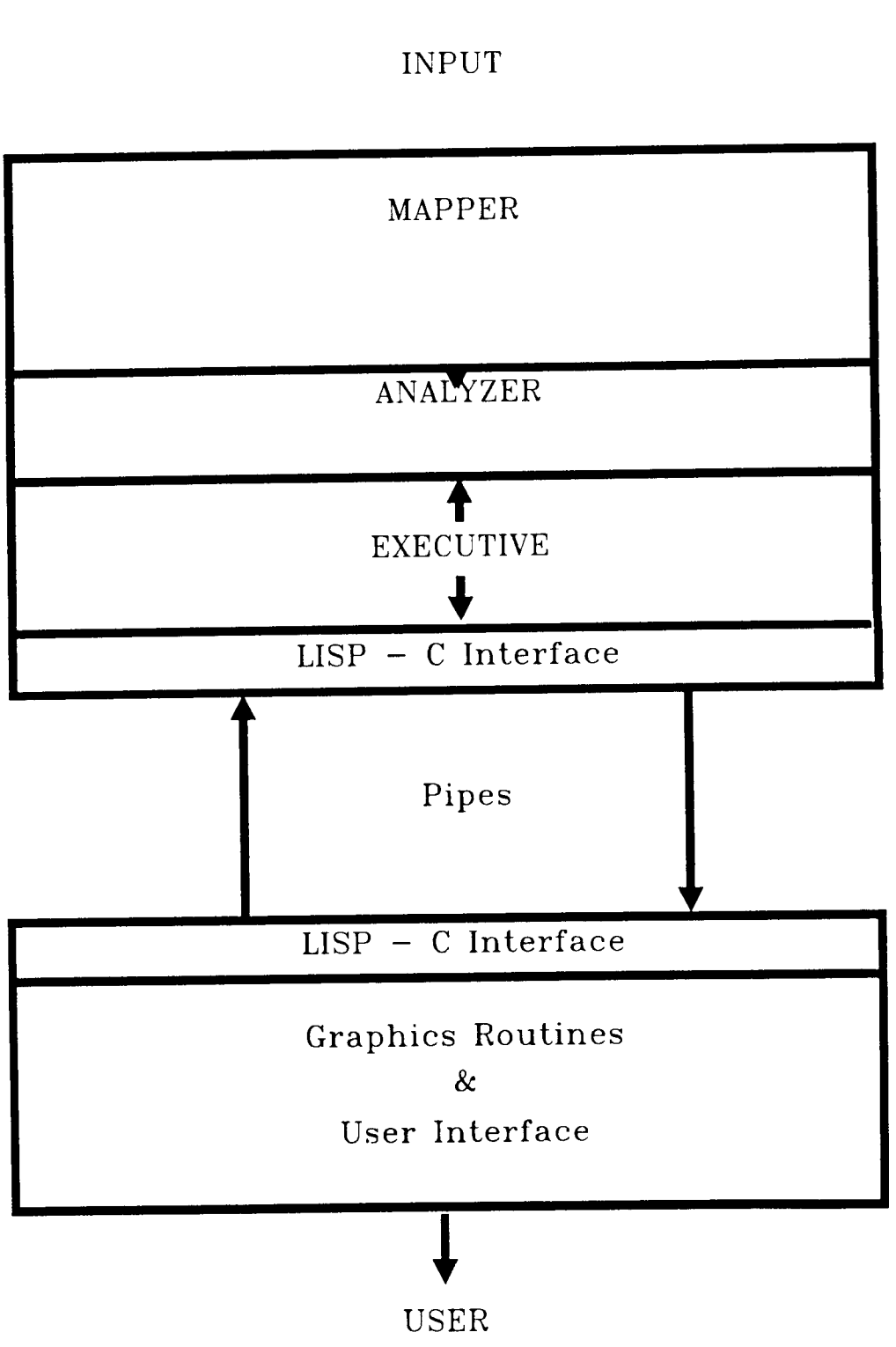

Figure 2 : Structure of Implementation

testing of large object-oriented systems.

Currently the facilities provided by the TVE are:

1. Generating the inheritance hierarchy for the entire system.

2. Generation of the inheritance tree for specific object classes in the system.

3. Detection and highlighting of cyclic inheritance of object classes.

4. Detection of missing class and method definitions.

5. Detection of conflicting method definitions (i.e., an object inherits methods of the same name from two different classes, but these two are in no way connected, i.e., they lie on two different paths in the inheritance tree.)

The sequence of actions performed is as follows:

The MAPPER accepts the object-oriented system written in a particular object-oriented programming language as input, and maps it into a graph.

Each object class in the system is mapped onto a node in the graph, and edges are defined as follows:

If an object class A inherits(includes) the class definition of object class B , then there is an edge from the node representing class A to the node representing class B. With this simple algorithm the entire graph is built. The current system implements a mapper for a Flavors-like object-oriented system, and uses the same algorithm as Flavors to determine method inheritance. The difference is that all information is explicitly displayed to the knowledge engineer, before expensive dynamic testing takes place.

For example, in Flavors, cyclic dependencies of objects are avoided, but the knowledge engineer is not notified. In the TVE all cycles are explicitly displayed.

On building the entire graph, the mapper terminates, and control passes to the executive which creates and communicates with the graphics server.

The graphics server, on creation, generates a window for the user interface. This window consists of a panel of test options, and a large canvas for displaying the graph generated for the system. An additional text sub-window is created for display of textual information about the system (e.g., object definition or list of inherited methods).

The user can now select any test option by simply selecting that option from the panel with a pointing device. This selection is communicated to the executive who, in turn, invokes analyzer routines to carry out the test. Information about any object in the system is obtained simply by pointing at the corresponding node in the graph.

TVE has also been used for supporting the static analysis of large

rule-based systems. Specifically, it has been successfully tried on a rule base containing approximately one hundred OPS5 rules.

CONCLUSIONS

The purpose of this paper was to discuss some of the software engineering aspects of AI programming and to describe a method and corresponding tools developed for testing and validating knowledge bases. The essence of the method is that the criteria for correctness is expressed in the form of structural properties and checked by using various graph algorithms.

The conclusion of our analysis was that the result of the evaluation depends on the approach to AI programming. Testing and validation of certain AI systems which try to mimic manifestations of human intelligence (e.g., expert systems) may be quite problematic because of the inherent difficulties in specification and performance evaluation. On the other side, programming techniques which are generally used in AI programming (e.g., declarative programming, symbolic programming, etc.) offer new opportunities for testing and validating the "knowledge base" of complex systems. These opportunities serve as one of the main incentives to use AI programming techniques in the design and implementation of complex systems.

This conclusion is quite contradictory to the often emphasized view, that AI techniques are "unsafe" compared to conventional programming techniques. The fundamental feature of knowledge-based systems is that most of the complexity is concentrated in their knowledge base. The dominantly declarative character of knowledge bases allows the application of automatic testing and validation techniques that can significantly improve the safety and reliability of large software systems.

REFERENCES

Allmendinger, G., "AI: Can Performance Match the Promise?," InTech, pp. 45-50, April, 1986.

Gashing, J., et al., "Evaluation of Expert Systems," in Building Expert Systems, F. Hayes-Roth, D.A. Waterman, and D.B. Lenat, eds., Addison Wesley, 1983.

Lane, N.E., "Global Issues in Evaluation of Expert Systems," Proc. 1986 International Conference SMC, pp. 121-125, 1986.

Newell, A., "The Knowledge Level," Artificial Intelligence 1:87-127, 1982.

Nguyen, T.A., "Verifying Consistency of Production Systems," Proc. of The Third Conference on AI Applications, pp. 4-8, 1987.

Stachowitz, R.A., et al., "Validation of Knowledge-Based Systems," Second AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program, 1987.

Suwa, M., et al., "An Approach to Verifying Completeness and Consistency